Traditional computer systems are built around the solitary central processor—an omnipotent agent that executes instructions and commands peripheral devices. Traditional programs reflect this monolithic orientation; programs describe a single instruction stream, with instructions evaluated sequentially. It is now possible to build systems with many active computing agents—independent processors that execute concurrently but can nevertheless communicate. We need to develop new software technology to control such systems. In this book we explore some of the tools and techniques that have been proposed for programming systems composed of many independent agents and for focusing these systems on single tasks. We give the name *coordinated computing* to this study of organizing multiple, independent, and loosely connected computing systems to perform coherent problem solving.

Historical Perspective

The historical patterns of use and cost of computing systems have changed dramatically. Early computers were extremely expensive, physically large, and computationally slow. They were designed for use by a single programmer at a time, who had the entire machine devoted to his or her use. Such systems did not have an operating system to protect against malice or mistake. Instead, the user's program controlled the computer directly. Since computers were expensive, they were shared — one user at a time. Users "signed up" to reserve time on the computer.

Clearer understanding of computation, cheaper machines, and a desire for improved system utilization led to the batch/stream computer. Here the computer scheduled its work, running each user's task in turn. Typically, a programmer submitted a program on a deck of punch cards in the morning and returned later that afternoon for the output. Better systems provided two or three runs a day. Primitive operating systems were developed primarily to order the batch stream and to arrange for each program's tapes. In these systems, security was limited to ensuring that programs used the correct files and tapes. Since only

a single program was running at a time, programs did not interfere with each other's address spaces.

Interactive timesharing systems have replaced batch systems, at least in those environments devoted to program development. A timesharing computer is a complex system. Instead of sequencing the tasks of a series of users, it interleaves them. With timesharing, productivity increases dramatically. Timesharing systems provide facilities such as interactive database access and text editing. However, timesharing requires a more complicated operating system than batch. A timesharing system must provide each concurrent program with a secure address space of its own; the system must switch rapidly between user contexts. Timesharing is possible because of large, fast machines. With these machines, a fraction of the computer's resources is enough to accomplish a single user's work.

This is the age of microprocessors—computers so cheap that their processor cost has almost ceased to matter. Today, a few hundred dollars can purchase more computational power than millions could buy in the 1950s. We are seeing the beginning of the "personal computer" age, where every worker has a computer of his or her own. These machines are usually connected by a network that provides intermachine communication and shared data. In a sense, we are coming full circle: The computer is no longer a shared device, but is being returned to the individual user.

The trend toward cheaper, smaller, and faster machines continues. As fabrication techniques continue to improve, the single processor on a chip will give way to a phalanx of processors on a chip and an army of interconnected chips. We believe that the next generation of computer architectures will provide each user with not just one but many computers—the *megacomputer*. However, improved computational productivity is not achieved by processing power alone. Special problems arise in coordinating systems of independent, asynchronous agents. Along with multiple processor architectures must come the software facility to exploit that computing power. A *coordinated computing system* successfully integrates multiple processing elements for problem solving. This book is a compendium of ideas about the software issues involved in programming coordinated computing systems.

Building a timesharing system is a difficult task. Interleaving computations, arranging to switch contexts between programs, and ensuring the security of each individual process requires proficient engineering. Nevertheless, the problems of timesharing are well understood. Creating a timesharing system is no longer a research endeavor, but an engineering activity. We see the next intellectual step in the development of computing systems as that of harnessing the power of the megacomputer.

Coordinated Computing Systems

A coordinated computing system distributes the work of a single task among many processing agents. Building a coordinated system is much harder than

constructing a timesharing computer. Coordinated computing is like timesharing in that we must arrange to do many activities simultaneously. However, unlike timesharing, coordinated computing requires the ability to focus multiple, simultaneous activities on a single goal.

How does coordinated computing differ from conventional programming? To understand the differences we must make several distinctions. We need to distinguish processors and processes. A *processor* is a physical box that *computes*—executes the instructions of a program, moves pulses of electricity around wires, etc. Processors execute instructions. Processors use a fixed and permanent storage.

A *process* is a logical processor. Like a conventional processor, a process executes a program and possesses storage. Unlike a processor, a process is not tied to any particular physical object. Analogically, a process is to a processor as software is to hardware.*

Often systems associate several processes with a single processor. For example, in timesharing systems each user (or user task) gets a process. The timesharing system tries to make that process appear to be running on its own (almost) independent processor. Many of the systems that we describe in this book are based on the synchronization of and communication among independent processes.

Though processes must have some independence, they should not become too isolated. Processes must be able to communicate—to transfer information among themselves. After all, coordinated problem solving requires communication. One crucial dimension of communication is *bandwidth*—the amount of information that communicators can exchange in a given time. We classify multiple processor systems by their communications bandwidth. Systems that allow sharing of much information are *multiprocessors*. Such systems can be thought of as providing shared memory to their processes. We use the term *shared memory* to describe this close information coupling because such systems are usually implemented by sharing primary memory between the multiple processors. With shared memory, communication is inexpensive.

Only a limited set of architectures provide inexpensive communication. More generally, communication has its costs. Systems that incur higher communication costs are *distributed systems*. In this book, we focus on software techniques for controlling and exploiting distributed systems.

A final distinguishing attribute of coordinated computing is a requirement for coherent problem solving. We are not interested in just getting computers to communicate (the study of computer networks), nor are we interested in providing the software foundation for application program communication and synchronization (the study of distributed operating systems). We want our processes to cooperate in the partitioning and resolving of tasks. An appropriate technique

* Here we use "process" for what is conventionally called a "logical process" in operating systems.

for the study of coordinated computing would be case studies of such systems. Since distributed problem solving systems have not yet been built, we cannot follow that path. Instead, in this book we take various proposals for the appropriate organization of multiprocessor and distributed systems that have appeared in the scientific literature and develop their themes. We emphasize the theoretical organizing principles of these ideas instead of the engineering decisions of particular implementations.

Building a coordinated computing system involves two primary activities: constructing and connecting the system's hardware and programming the system's software. This book is about software. Clearly, developing hardware is crucial to building coordinated systems. Nevertheless (except for a few definitions and pointers to the literature), we virtually ignore hardware. Instead, we take the point of view that such physical systems will come into existence; the technology to build inexpensive processors and to get them to communicate already exists. We are interested in the effective use of these emerging systems—programs that can use a coordinated system as something more than a complicated sequential processor. This book is an investigation of possible alternatives for constructing coherent multiprocess systems.

Models, Languages, and Heuristics

We believe that building coordinated computing systems requires understanding of three different facets of system organization: models, languages, and heuristics.

Models capture the abstract relationships between the important components of systems. To evaluate a system, one must know the parameters of its construction: how long particular instructions take to execute, the effects of specific statements, and so forth. Modeling is particularly important when considering emerging technologies. Such technologies need models both to guide the system development process and to substitute for observations of system performance. Models are used in system design, validation, and analysis. In Part 2 we discuss several models that apply to the problems of coordinated computing.

The usual way to give directions to a computer is with a *program* written in some *programming language*. Coordinated systems need programming languages that can describe concurrent activity and communication. Some of our experiences with traditional programming systems are an impediment to designing languages for coordinated systems. Traditional programs are executed sequentially. Their control structures can specify only serial activities: "First, do this; next, do that." The primary advantage of coordinated systems is the increased processing power of concurrent computation. However, if system components are to execute concurrently, then they must be able to determine the (potentially) concurrent activities. In general, this can be done in one of two ways: either (1) the programmer can indicate parallel actions with specific programming language constructs, or (2) the system can infer opportunities for parallelism on its own.

The programs one writes reflect the facilities of one's programming system. Classical sequential programming languages (such as Pascal, Cobol, and Lisp) are inadequate for programming coordinated systems. These languages do not treat important problems such as concurrency, communication, synchronization, security, and failure. In Part 3, we consider several different language proposals that address some of our requirements. These languages are primarily *distributed languages*—concurrent languages that recognize the cost of communication.*

Typically, models address the formal, mathematical understanding of systems, while programming languages mediate directions to a computer. Programming languages have a complete syntax and semantics. Models usually express only the simplest of relationships between a system's elements. The systems we consider are usually describing the control of decentralized computer systems. Hence, many systems blend elements of language and model, often taking the form of a few additional concepts to be added to a standard language like Pascal or Lisp.

Programming in sequential languages has taught us about sequential solutions to problems. Coordinated systems provide the opportunity to program concurrent solutions. However, except for the simplest cases, the exercising of such concurrent power is an intellectually demanding task. This is especially true when there are many active processes. *Heuristics* for coordinated computing are ideas on the "organizational" or "social" architecture of coordinated systems—techniques for getting processes to work together and for exploiting potential concurrency. In Part 4 we discuss heuristic organizations for coordinated problem solving systems.

Motivations for Coordinated Computing

This book promotes the idea of coordinated computing. Clearly, organizing a distributed, asynchronous system is harder than organizing one that is centralized or synchronous. So why bother? There are two major motivations for studying coordinated computing: economic and intellectual. On one hand, taming concurrent computation promises virtually cost-free processing. The massive amount of computing cycles that a coordinated system will provide will make many currently intractable computational problems solvable. (These include problems drawn from domains such as large system simulation, database access, optimization, and heuristic search.) On the other hand, organizing and understanding a

* We distinguish distributed languages and concurrent languages. Concurrent languages assume concurrently executing processes. However, these processes share storage (can communicate cheaply). Concurrent programming languages are better understood, more specific, and, in our opinion, less interesting than distributed languages. In Chapter 13 we discuss a concurrent programming language, Concurrent Pascal. We include this language for both historical and pedagogical reasons. Concurrent programming languages are also called multiprocessing languages.

set of independent agents is a challenging intellectual task. We find this combination of intellectual challenge and economic reward a compelling argument for the relevance of studying coordinated computing.

Book Overview

Our original title for this book was "Models, Languages, and Heuristics for Distributed Computing." In the course of our research, we came to the conclusion that there was some "whole" of distributed control greater than these three parts. We chose to call that whole *coordinated computing*. Nevertheless, our book structure still reflects our original triad. This book has five parts: "Foundations," "Models," "Languages," "Heuristics," and "Contrasts and Comparisons." The first part, Foundations, covers the minimal required background material and definitions. The next three parts survey proposed models (Part 2), languages (Part 3), and heuristic organizations (Part 4) for computation that we feel bear on coordinated computing. We compare and contrast these models, languages, and heuristics in Part 5, presenting a taxonomy of systems.

Having neither a general theory of coordinated computing nor a large pool of implementation experience, we chose to approach the problem by discussing relevant ideas from the computing literature. These ideas center on programming languages. However, the discussion in the rest of the book touches on many fields besides programming languages (and mentions some programming language concepts that may not be familiar to every reader). We therefore devote the first four chapters to developing background material: Chapter 1, Computation Theory (automata theory, lambda calculus, and the analysis of algorithms); Chapter 2, Programming Languages (syntax and semantics, and pragmatics); Chapter 3, Concurrency (concurrency, resource conflict, and synchronization); and Chapter 4, Hardware. Our intention is that the reader only marginally acquainted with a subject can, by reading the introductory section, learn all he or she needs to know to understand the rest of the book.

The last chapter of Part 1, Chapter 5, forms the introduction to Parts 2, 3, and 4. It outlines the nature of models, languages, and heuristics and touches on some of the difficulties faced by concurrent and distributed systems. This chapter introduces the dimensions of distribution by which we classify the various systems.

Part 2 surveys models for coordinated computing. Each chapter (6 through 12) in that part describes a different model (or a related set of models). For each, we describe the model and present several examples of its use. Part 3 (Chapters 13 through 16) is a similar survey of programming languages.

Part 4 discusses heuristics for organizing coordinated computing systems. Its first chapter, Chapter 17, discusses algorithms for distributed databases. Its other chapter, Chapter 18, develops some of the more interesting ideas for organizing distributed systems for coherent problem solving. Though much of

the work described in that chapter has its roots in artificial intelligence research, no particular background in that field is needed to understand the material.

Our final part, Part 5, contrasts and compares these systems, both in terms of the dimensions outlined in Chapter 5, and when appropriate, by similar and contrasting features. We conclude with a section discussing the characteristics of basis and ideal systems.

We tried to write the chapters in Parts 2, 3, and 4 so that each is (by and large) conceptually independent. While this independence is not complete, we feel that readers will be able to read just those chapters that interest them. More specifically, the reader interested in just one system [for example, Communicating Sequential Processes (Chapter 10) or tasking in Ada (Chapter 14)] can skip to that chapter; the reader who finds a chapter too difficult [as many not familiar with the lambda calculus may find Concurrent Processes (Chapter 8)] can omit that chapter at first reading.

Every chapter includes a few exercises. These exercises form three classes: (1) brief *mention* exercises that draw the reader's attention to a tricky point in one of the examples, (2) *homework* problems that request the straightforward programming of a conventional problem in a new system, and (3) *research* exercises that describe a difficult problem. Some of these exercises are suitable for term projects; others are open research questions. Problems of this last type are marked by a "†". Each chapter ends with a bibliography of relevant papers. We have annotated those references when appropriate. There is a cumulative bibliography at the end of the book.

Audience

We have tried to write this book so that it can be understood by someone acquainted with the construction of programming languages—roughly the material in a junior level course on programming languages. A reading knowledge of Pascal (or the equivalent) is essential; a reading knowledge of Lisp is useful for understanding certain sections. The mathematical sophistication of the junior-level computer science student is also required at times. We have tried to avoid demanding a greater background of the reader. However, this material ranges over a wide territory—programming languages, operating systems, database systems, artificial intelligence, complexity theory, and computational theory. We attempt to explain, briefly, each potentially unfamiliar idea and to provide references to more complete descriptions. We urge the reader who finds him or herself in a familiar section to skip to more challenging material. In particular, much of Part 1 will be familiar to many readers.

We anticipate two audiences for this material. The first is the academic community. We use this book as the text in a graduate seminar on concepts of programming languages and include some material (particularly exercises) specifically for classroom use. The second audience is professional programmers.

We believe that coordinated computing will come to have profound economic importance. We have searched the scientific literature for important ideas applicable to coordinated computing and have expressed those ideas in an accessible form. We hope that this volume proves to be a sourcebook of ideas for the people who will actually develop coordinated computing systems.

Instructional use

We use this book as the text for a graduate seminar on advanced concepts of programming languages at Indiana University. Our approach is first to develop the concept and implementation of a process scheduler and then to introduce concurrency. We proceed to discuss most of the systems, describing the material in Part 1 as needed.

In that class, the term project is to implement the important semantic aspects of a distributed model or language. The students build their chosen language or model in Scheme [Steele 78]. We use Scheme because its powerful core and extensible nature make it well-suited for language design and implementation. Several Indiana University technical reports describe particular student projects ([Wolynes 80, Dwyer 81]).

Most classes will not be able to cover the entire book in a single term. In our opinion, every class should read this Preface, Chapter 5, and Chapter 19. The instructor should select a representative set of the important systems, covering those systems and the material in Part 1 needed to understand them. For example, one curriculum would include Shared Variables (Chapter 6), Exchange Functions (Chapter 7), Communicating Sequential Processes (Chapter 10), Actors (Chapter 11), Ada (Chapter 14), PLITS (Chapter 15), and the heuristic material in Part 4.

Structural Choices

In writing a book that covers such a broad territory we made many choices about which material to include and how to present it. We know that some of these choices will displease some people; clearly, we could have emphasized different aspects of our subject or described it differently. We have been driven by an interest in (1) the organizational requirements of coherent distributed computing, and, more particularly, (2) the underlying run-time structure of our various systems. We have deliberately avoided providing either formal semantics or correctness proofs. Though such formality has its research virtues, we feel that it would obscure the content of the book for most readers.

Another choice we faced was whether to preserve the original languages of the systems or to invent a new language, describing the systems in Parts 2, 3, and 4 in that language. We chose (by and large) to keep the originals for the following reasons: (1) By seeing the original language, the reader can

get a sense of the real structure and pragmatics of each system; (2) The reader who is familiar with the original can pursue that system in the literature without having to translate mentally to a new language. We made exceptions to this rule when the original model did not have a language, the full language was too obscure, or the particular language was undergoing rapid revision. In all such cases we invented an appropriate syntax to describe the system. One of our current research interests is a tractable universal language that can adequately describe the operational behavior of all these systems.

We use several examples in the description of each system. We might have selected a common example (or set of examples) to be used throughout the book. Instead, we vary the examples but use some repeatedly. We made this decision because the systems cover a wide range of facilities; an interesting example for a model is often too low-level an example for a programming language, while an interesting program is often far too complicated to express in most models. Instead, we have a common set of base examples and use some of these examples (and some others) in each section. Since most of these systems are theoretical, we have not been able to debug the programs on implementations.

Our apologies go to those system designers whose systems have been omitted. We have not been trying to write an encyclopedia of distributed systems. Instead, we selected those systems we feel are representative or important and described them in depth. This has, of course, meant that many systems have been left out. Some of these systems are briefly described in the bibliographic annotations of the appropriate chapters.

Acknowledgments

The help of many individuals and organizations has been important in completing this book. We thank Greg Andrews, David Bein, Gary Brooks, Jim Burns, Will Clinger, Gray Clossman, Dan Corkill, Jack Dennis, Scot Drysdale, Jerry Feldman, David Gries, Cordy Hall, Chris Haynes, Carl Hewitt, Tony Hoare, Eugene Kohlbecker, Steve Johnson, John Lamping, Bob Leichner, Egon Loebner, Barbara Liskov, John Lowrance, Nancy Lynch, George Milne, Robin Milner, Fanya Montalvo, John Nienart, Ed Robertson, Vlad Rutenberg, Rich Salter, Bob Scheifler, Avi Silberschatz, Mitch Wand, Peter Wegner, David Wise, and Pam Zave for discussions and comments. We would particularly like to thank Steve Muchnick and Peter Thiesen for their comprehensive comments. This book is much easier to understand and more accurate for the help of these people.

A paper by David MacQueen [MacQueen 79] originally inspired our interest in the subject of distributed models and languages. This interest led to our teaching a seminar. The ideas of that seminar evolved into the concept of coordinated computing and this book.

Several people and organizations have allowed us to quote or adapt their previously-published material. We would like to thank:

Prentice-Hall for permission to adapt Figure 4-1 from Figure 1-5, p. 11 of *Computer Networks* by Andrew S. Tanenbaum, Copyright 1981. Adapted by permission of Prentice-Hall, Inc. Englewood Cliffs, N.J.

Prentice-Hall for permission to adapt Figure 9-6 from *Petri Net Theory and the Modeling of Systems*, by James L. Peterson, p. 67, Copyright 1981. Adapted by permission of Prentice-Hall, Inc. Englewood Cliffs, N.J.

Jack Dennis and Springer-Verlag for permission to adapt Figures 9-8 through 9-14 and 9-16 from Figures 1 through 5 of the article "First Version of a Data Flow Language," by Jack Dennis in *Proceedings, Colloque sur la Programmation*, B. Robinet, ed., Lecture Notes in Computer Science vol. 19, Copyright 1974, Springer-Verlag.

The ACM for permission to adapt Figures 8-1 through 8-5 and 8-9 from the article "Concurrent Processes and Their Syntax," by George Milne and Robin Milner, *JACM*, vol. 26, no 2, April 1979, Copyright 1979, Association for Computing Machinery, Inc., reprinted by permission.

Steven D. Johnson for permission to adapt Figures 12-5 and 12-11 from *Circuits and Systems: Implementing Communication with Streams*, TR 116, Computer Science Department, Indiana University.

Springer-Verlag for permission to adapt Figures 12-6 through 12-10 from Figures 8 and 10 of the article "An Approach to Fair Applicative Multiprogramming" by Daniel P. Friedman and David S. Wise in *The Proceedings of the International Symposium of Concurrent Computation*, Gilles Kahn, ed., Lecture Notes in Computer Science vol. 70, Copyright 1979, Springer-Verlag.

Daniel Corkill for permission to reprint the quote in Exercise 18-1 from *Cooperative Distributed Problem Solving: A New Approach for Structuring Distributed Systems*, by Victor Lesser and Daniel Corkill, TR 78-7, Department of Computer and Information Science, University of Massachusetts, May, 1978.

The ACM for permission to reprint the quote in Chapter 19 from "High level programming for distributed computing," by Jerome Feldman, *Communications of the ACM*, vol. 22, no. 6 (June 1979), Copyright 1979, Association for Computing Machinery, Inc, reprinted by permission.

We thank the Computer Science Department of Indiana University and the Computer Research Center of Hewlett-Packard Laboratories for their support and the use of their facilities. We also thank the National Science Foundation for its support over the years, and Charles Smith and the System Development Foundation for support for a California visit to complete this project.

Myrna Filman, Kim Fletcher, Peg Fletcher, and Nancy Garrett have provided administrative, organizational, and artistic help. The output of a program by John Lamping was the basis of the jacket design. We typeset this book at Stanford University using Don Knuth's TEX system. The expertise of Bob Ballance, David Fuchs, and Rich Pattis greatly facilitated the typesetting process. They all have our gratitude.

And finally, our love and thanks to our wives, Myrna Filman and Mary Friedman, whose emotional support and understanding were crucial for a project of this magnitude.

Robert E. Filman
Daniel P. Friedman